A Stream-Computing Extension to OpenMP

Antoniu Pop[1] and Albert Cohen[2]
[1] Centre de Recherche en Informatique, MINES ParisTech, France
[2] INRIA Saclay and LRI, Paris-Sud 11 University, France

1 Motivation and design

Optimizing compilers and runtime libraries no longer shield programmers from the complexity of processor architectures. As a result, the efficient use of multi-core hardware increasingly relies on manual and target-specific optimization. This trend drives the design of high-level languages that allow programmers to expose concurrency and data locality properties with little dependence on a target architecture. The stream-programming model is a family of such languages: it decomposes programs into tasks and makes the flow of data among tasks explicit. This model exposes data, task and pipeline parallelism. It improves locality and the ability for the compiler to adapt the concurrency to a given target; it also helps the programmers to avoid non-portable coding practices and early optimizations restricting parallelization opportunities. Our work is motivated by the strong evidence that has been gathered on the importance of leveraging pipeline parallelism in order to achieve efficiency and scalability, without losing productivity.

The current OpenMP API lacks the capability to explicit the data-flow between tasks. The existing sharing clauses only allow to distinguish between shared and private data and to provide, with firstprivate, initialization values for task private data. In order to use task constructs in non-embarrassingly parallel problems, manual synchronization is required.

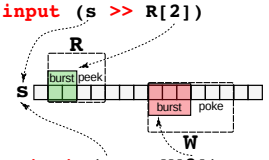
Our primary design goal is to enable OpenMP programmers to exploit pipeline parallelism without explicitly having to handle communication and synchronization, which is both error-prone and time-consuming. We also want to offer highly efficient decoupled pipelined executions to programmers with no experience in shared-memory concurrency. To achieve these goals, we propose minimal and incremental additions to the OpenMP language, exposing the producer-consumer relationships between tasks and enabling the generation of pipelined parallel code, while ensuring this additional expressiveness does not introduce excessive complexity and does not break the semantics of the current standard.

2 Proposed streaming extension

Language extension. We propose to extend the OpenMP3.0 standard with two additional clauses for task constructs, the input and output clauses presented on Figure 1, as well as to modify the current execution model of OpenMP tasks, making *streaming* tasks persistent.

```
input/output (list)
    list   ::= list, item
             | item
    item   ::= stream
             | stream >> stream
             | stream << stream
    stream ::= var
             | array[expr]
    expr   ::= var
             | value
```

```
int s, R[4], W[6];
input (s >> R[2])
```




```
output (s << W[3])
```

Figure 1: Syntax for input and output clauses.

Both clauses take a list of items, each of which describe a stream and its behaviour with regards to the task to which the clause applies. If the item notation is in the abbreviated form `stream`, then the stream can only be accessed one element at a time through the same variable stream. In the second form, `stream>>stream`, the programmer uses the C++-flavoured `<< >>` stream operators to connect a sliding window to a stream, gaining access, within the body of the task, to *horizon* elements in the stream.

Our programming model is more general than data-flow: tasks compute on streams of values and not on individual values. An array declaration (in plain C) defines the sliding window accessible within the task. Streams are implicit from the matching of input and output clauses, which also define the displacement of the sliding windows in their respective streams for each activation. The data-flow case corresponds to $horizon = burst$. In the more genaral case where $horizon > burst$, the window elements beyond the burst are accessible to the task; for an output window (output), the values of these elements will only be committed in subsequent activations and made accessible to consumers. Task activation is driven by the availability, on each input stream, of all elements of the input windows, including the elements beyond the burst.

The exammples on Figure 2 illustrate the syntax of the input and output clauses. In the first example on the left, the task reads from the stream x, reading up to horizon values of x ahead of the current position in the stream and consumes burst elements at each activation. In the second example on the left, the task reads from the stream A[0], the first element of the array of streams A, and renames it to z for use within the task. As the right hand side of the clause item is not an array, the task can only access and consume one element at a time. The third example on the left, the task from the stream of arrays A of 3 elements; depending on the task, the same array may be used as an array of streams or a stream of arrays. Eventually, the third example on the right shows

a stream of arrays with parametric horizon and burst values; arbitrary combinations are possible.[1]

```
int x, z;
int X[horizon];
int A[3];

#pragma omp task input (x->X[burst])
  // task code block
  ... = ... X[2]; // with horizon > 2

#pragma omp task input (A[0]->z)
  // task code block
  ... = ... z ...;

#pragma omp task input (A)
  // task code block
  ... = A[0] + A[1] + A[2];
```

```
int y;
int Y[horizon];
int B[horizon][2];

#pragma omp task output (y)
  // task code block
  y = ...

#pragma omp task \
        output (y<-B[burst][])
  // task code block
  for (int i=0; i<burst; ++i) {
    B[i][0] = ...;
    B[i][1] = ...;
}
```

Figure 2: Examples of input and output clause uses.

Execution model. The OpenMP3.0 task execution model is very similar to that of coroutines or fibers. Tasks may either be scheduled immediately on the same thread or deferred and assigned to any thread in the team; in the general case, no ordering, no exclusion and no thread locality can be assumed. Such an execution model is well suited for very unbalanced loads or when the target architecture offers some support for very lightweight threading, but in most cases the overhead of creating and scheduling the tasks is significantly more expensive than synchronizing persistent tasks.

We propose to change the execution model to make streaming tasks persistent. We emphasize the fact that we only modify the execution model for streaming tasks: the semantics of OpenMP programs is not impacted. This choice puts a heavier load on the compiler: it needs to convert the dynamic scheduling of new instances of a task into data-driven synchronization (i.e., based on the availability of data in the input streams).

Correctness is of course the burning question at this point. Overall, the transformation is always possible and correct when the only scheduling constraints are the data-driven synchronizations enforced by input and output clauses. Obviously, introducing atomic sections within tasks is compatible with any data-driven synchronization constraint. However serious causality problems may arise when combining our streaming extensions with arbitrary locking mechanisms, if the acquisition of a lock escapes outside task boundaries. In real applications, locking may be legitimate to handle other forms of concurrency unrelated with the parallelization itself (e.g., I/O or user interfaces).

3 Implementation and experiments

The implementation of this extension is under way in a public branch of GCC. While this is still an early implementation that does not provide full support for the OpenMP streaming extension, it has reached the point where programs requiring only simple pipelines, and no mixed data/pipeline-parallelism, can be compiled. The current version only supports the simplified syntax, therefore restricting burst and horizon sizes.

[1]The size of the second dimension of B can be implicit in the clause.

We present results on three full applications: `FFT` from the StreamIt benchmarks [?], `FMradio` from the GNU radio package and also available in the StreamIt benchmarks[2] and a `802.11a` production code from Nokia[3]. These applications are complex enough to illustrate the expressiveness of this extension.

The fully automated stream code generation from OpenMP, with our extension, was only sufficient for `FMradio` and `802.11a` and only exploits pipeline parallelism. We achieve more than 3x speedup on `FMradio` and 1.9x speedup on `802.11a` on an Intel Core2 Quad Q9550 with 4 cores at 2.83GHz.

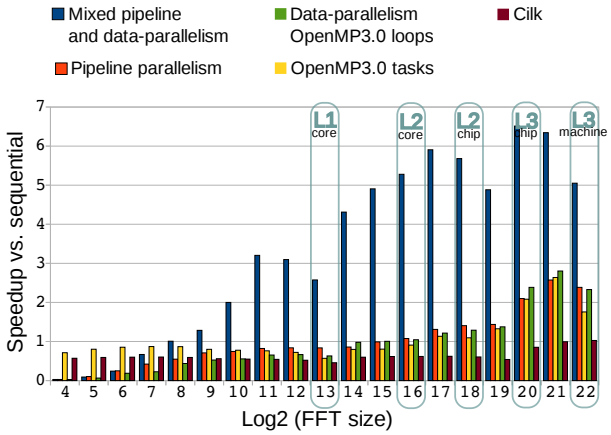


Figure 3: FFT performance comparison on Opteron.

In order to give an idea of what could be achieved once the implementation is complete, we hand-streamized the three applications and measured the performance on a 4-socket AMD quad-core Opteron 8380 with 16 cores at 2.5GHz and 64GB of memory. `FMradio` presents a high amount of data-parallelism and is fairly well-balanced; it gets up to 14.6x speedup. `802.11a` is more unbalanced and only achieves up to 7.45x speedup. The speedups on `FFT` are presented on Figure 3. The baseline is an optimized sequential FFT implementation used as a baseline for the StreamIt benchmark suite. Combined task- and data-parallelism achieve the best speedups, compared to pure data-parallelism or pure pipelining. The size of the machines and the associated cost of inter-processor communication sets the break-even point towards vectors of 256 double floating point values and more.

4 Conclusion

We presented an incremental extension to enable stream programming in OpenMP. Our work is motivated by the quest for increased productivity in parallel programming, and by the strong evidence that has been gathered on the importance of pipeline parallelism for scalability and efficiency.

[2]`http://gnuradio.org/trac`

[3]From the ACOTES FP6 European project.